The Midori OS & Its Programming Language

a.k.a. Midori C#, a.k.a. M#

Joe Duffy, Jared Parsons, Aleks Bromfield

What is Midori?

- Systems incubation started in 2007 to solve hard problems w/ a “no legacy” mindset.
    - New OS built from the ground up.
    - Managed code safety; native code performance.
    - Strong isolation. “No lies” resource management. No paging.
    - Non-negotiable type, memory, concurrency-safety. Including drivers.
    - Non-blocking. Inherently concurrent. No hangs. Race-free parallelism.
    - Get ahead of disruptive trends. Heterogeneity. Run natively on x86, x64, ARM T2.

- Unique culture.
    - All developers.
    - Everybody codes, everybody loves to code.
    - We build the system by building the system.
    - Strong engineering culture; 100% test coverage in many cases.

- All user-facing code, and 99% of Midori system code (4 MLOC), written in Midori C#:
    - Kernel, GC, NetStack, FileSystem, Drivers (USB, SATA, 10GNIC, etc), HTML5/JS Browser, WebServer, Multimedia, etc.
    - Exceptions: Policy-free C++ Microkernel, Redhawk GC, Chakra JS Engine.

What is Midori C#?

- Midori started with Sing#, and then C#.
  - Added on top rigorous static analysis and fail-fast discipline:
    - No Reflection / Dynamic.
    - All standard FxCop rules, 100 custom rules.
    - Optimized ahead-of-time compilation via Bartok and Phoenix.
    - But soon reached the limits of a “hands off the language” approach.

- I joined in 2009; kicked off Midori C# in 2010.
  - Based on pre-Roslyn CSC.EXE code-base.
  - Incremental improvements at first. E.g., ‘awaits’ before there was an awaits.
  - Slowly innovated more, e.g. immutability, but stuck to “additive & compatible” mantra.
  - Recently changed our mindset: optimize for new code, not porting existing code.

- We try to mimic the C# Design Team as much as possible.
  - Developers: Aleks Bromfield, Jared Parsons, Joe Duffy, Shon Katzenberger
  - Design Team: All of the above, Adam Nathan, Krzysztof Cwalina, Matt Ellis, Ryan Zelen

Midori Status

- Midori is running in production.
  - In Dec 2011, SteveB and Qi asked us to get a “real” Bing/Midori workload in production.
  - Partnered with IPE Speech Recognition Team. 500KLOC of C++ became 300KLOC of Midori C#.
  - First production request served up in Dec 2012. Now 15% of Phone traffic. “>20 million served.”
- Results are fantastic!
  - 0.5s – 1.5s less latency / request. 3x throughput. All safe code!
  - Actively working towards 100% in 2013. Xbox. DNN. Branching out from there.
- Despite server focus, still general purpose OS: ARM, laptop, UI, Browser. NT circa 1992.
- Starting to track to a Midori V1; goal is to stabilize Midori C# mid-to-late 2013.

Language/Talk Overview

- Taming Side-Effects (TSE): immutability, isolation.
- Systems Programming: fast delegates, interfaces; stack allocation.
- Error Model: contracts; exceptions.
- A number of small features we won't discuss.

Principles and Goals

- The type system knows about all side-effects.
- No mutable statics. All capability-based authority.
- Goal is to enable:
    - Safe concurrency.
    - Fewer “surprises” ⇒ more correct code.
    - Functional code where desirable; imperative where needed.
    - Better semantic understanding and compiler analysis.
- All of Midori compiles w/ TSE on; illegal to opt-out, so compiler can trust.

The Basic Idea: Permissions

- References, values, and methods are annotated with permissions.

`writable Foo a = …;`	Object graph can be mutated **(default if unstated)**
`readable Foo b = …;`	Object graph cannot be mutated (by me)
`immutable Foo c = …;`	Object graph is frozen until the end of time
`P Foo d = …;`	Generic permission (advanced; more later)

- Modeled as subtypes; behavior falls out naturally (overrides, etc.).
  - **`writable`** <: **`readable`**
  - **`immutable`** <: **`readable`**

- **`readable`** `object` is the "top type" of the system.

Permissions on Methods and References

```
class Person
{
    public void SetName(string name) { … }

    public string GetName() readable
    {



    }
}

readable Person p1 = …;
```

Permissions on Methods and References

```
class Person
{
    public void SetName(string name) { … }

    public string GetName() readable
    {
        m_name = "SteveB";    // error CS7003: Assignment to target requires 'writable' permissions; you have 'readable'
        SetName("BillG");     // error CS7000: Calling 'Person.SetName(string)' requires 'writable' permissions; you have 'readable'
        return m_name;
    }
}

readable Person p1 = …;
```

Permissions and Generics

```
/// Deeply mutable object graph

List<Person> list1 = new List<Person>();
```

Permissions and Generics

```
/// Deeply mutable object graph

List<Person> list1 = new List<Person>();
list1.Add(new Person());
list1[0].SetName(…);
```

Permissions and Generics

```
/// Deeply mutable object graph

List<Person> list1 = new List<Person>();
list1.Add(new Person());
list1[0].SetName(…);

/// Shallow readable: list contents can't changed but elements can

readable List<writable Person> list2 = list1;
```

Permissions and Generics

```
/// Deeply mutable object graph

List<Person> list1 = new List<Person>();
list1.Add(new Person());
list1[0].SetName(…);

/// Shallow readable: list contents can't changed but elements can

readable List<writable Person> list2 = list1;
list2.Add(new Person());    // error CS7000: Calling 'List<T>.Add(T)' requires 'writable' permissions; you have 'readable'
list2[0].SetName(…);        // OK.
```

Permissions and Generics

```
/// Deeply mutable object graph

List<Person> list1 = new List<Person>();
list1.Add(new Person());
list1[0].SetName(…);

/// Shallow readable: list contents can't changed but elements can

readable List<writable Person> list2 = list1;
list2.Add(new Person());    // error CS7000: Calling 'List<T>.Add(T)' requires 'writable' permissions; you have 'readable'
list2[0].SetName(…);        // OK.

/// Deeply readable: list or contents can't be mutated in any way

readable List<Person> list3 = list2;
```

Permissions and Generics

```
/// Deeply mutable object graph

List<Person> list1 = new List<Person>();
list1.Add(new Person());
list1[0].SetName(…);

/// Shallow readable: list contents can't changed but elements can

readable List<writable Person> list2 = list1;
list2.Add(new Person());     // error CS7000: Calling 'List<T>.Add(T)' requires 'writable' permissions; you have 'readable'
list2[0].SetName(…);         // OK.

/// Deeply readable: list or contents can't be mutated in any way

readable List<Person> list3 = list2;
list3.Add(new Person());     // error CS7000: Calling 'List<T>.Add(T)' requires 'writable' permissions; you have 'readable'
list3[0].SetName(…);         // error CS7000: Calling 'Person.SetName(string)' requires 'writable' permissions; you have 'readable'
```

Case Study: Collections

- View based collection types no longer needed (ReadOnlyCollection).
- Often replaced with readable / immutable arrays (with efficiency gains).

```
class Map<contains TKey, contains TValue>
  : IEnumerable<KeyValuePair<TKey, TValue>>
{
  int Count { readable get; }
  IEnumerable<TKey> Keys { readable get; }
  IEnumerable<TValue> Values { readable get; }
  bool TryGetValue(readable TKey key, out TValue value) readable;
  bool Contains(readable TKey key) readable;
  void Add(TKey key, TValue value);
  bool Remove(readable TKey key);
}
```

Permissions and Generics

```
/// Deeply mutable object graph

List<Person> list1 = new List<Person>();
list1.Add(new Person());
list1[0].SetName(…);

/// Shallow readable: list contents can't changed but elements can

readable List<writable Person> list2 = list1;
list2.Add(new Person());    // error CS7000: Calling 'List<T>.Add(T)' requires 'writable' permissions; you have 'readable'
list2[0].SetName(…);        // OK.

/// Deeply readable: list or contents can't be mutated in any way

readable List<Person> list3 = list2;
list3.Add(new Person());    // error CS7000: Calling 'List<T>.Add(T)' requires 'writable' permissions; you have 'readable'
list3[0].SetName(…);        // error CS7000: Calling 'Person.SetName(string)' requires 'writable' permissions; you have 'readable'
```

Case Study: Collections

- View based collection types no longer needed (ReadOnlyCollection).
- Often replaced with readable / immutable arrays (with efficiency gains).

```
class Map<contains TKey, contains TValue>
  : IEnumerable<KeyValuePair<TKey, TValue>>
{
  int Count { readable get; }
  IEnumerable<TKey> Keys { readable get; }
  IEnumerable<TValue> Values { readable get; }
  bool TryGetValue(readable TKey key, out TValue value) readable;
  bool Contains(readable TKey key) readable;
  void Add(TKey key, TValue value);
  bool Remove(readable TKey key);
}
```

Immutability

- No mutable statics means “fresh” expressions are immutable.

```
immutable Person person = new Person() { Name = “Someone“, Age = 42 };

immutable List<Person> people = new List<Person>(
    new[] { person, … });
```

- Sometimes more complex initialization is required:

```
immutable Person person = Immutable.Create(() => {
    var p = new Person();
    … multiple initialization steps …;
    return p;
});
```

Isolation

- An isolated is a self-contained object graph.
  - No external references.
  - Non-aliasable references; linear usage.
  - Passing immutable data into and out of the graph is okay.
  - Immutability, subtly, is built on top of isolation.

- Two keywords: **isolated** and **consume**.

```
isolated List<Person> peopleBuilder = new List<Person>();
personBuilder.Add(new Person() { … });
personBuilder.Add(new Person() { … });

immutable List<Person> people = consume peopleBuilder;

// ‘peopleBuilder’ is now unusable.
```

Immutability

- No mutable statics means “fresh” expressions are immutable.

```
immutable Person person = new Person() { Name = “Someone“, Age = 42 };

immutable List<Person> people = new List<Person>(
    new[] { person, … });
```

- Sometimes more complex initialization is required:

```
immutable Person person = Immutable.Create(() => {
    var p = new Person();
    … multiple initialization steps …;
    return p;
});
```

Isolation

- An isolated is a self-contained object graph.
  - No external references.
  - Non-aliasable references; linear usage.
  - Passing immutable data into and out of the graph is okay.
  - Immutability, subtly, is built on top of isolation.

- Two keywords: **isolated** and **consume**.

```
isolated List<Person> peopleBuilder = new List<Person>();
personBuilder.Add(new Person() { … });
personBuilder.Add(new Person() { … });

immutable List<Person> people = consume peopleBuilder;

// 'peopleBuilder' is now unusable.
```

Delegates and Lambdas

- A delegate can have a “capture” permission associated with it.

```
public delegate TResult PureFunc<TResult>() immutable;
```

- Used to enforce policy; e.g., parallel code is race-free, LINQ queries don’t mutate, transactions can be retried, etc.

```
public static class AsyncFactory {
    public static Task<TResult> Run<TResult>(PureFunc<TResult> func);
}
```

- Now callers of Run cannot capture any mutable state. readable is possible too, less restrictive.

```
List<int> a = …;
readonly immutable List<int> b = …;
var tsk1 = AsyncFactory.Run(() => {
    int x = a[0]; // error CS7006: 'a' cannot be captured by delegate, because it is not readonly and immutable
    int y = b[0]; // OK.
});
```

Experiences Applying TSE to Midori

- Turned on TSE project by project.
  - Annotations per line of code is low… (but more code can benefit)
  - Took roughly 1 ½ years to port all 4 MLOC to TSE; iterated as we went.
- Found bugs! Primarily accidental mutations ("mutable immutability ☺").
- A few recurring pain points.
  - Generics took 4 tries to get right!
  - Mutable statics. Solution: use capabilities properly.
  - Side-effects in contracts, LINQ queries, etc. Solution: stop doing it!
  - Developer expectations vs. reality. Solution: fix the code
- Happy with the results; but we aren't done.
  - Big notable win recently: Frozen Objects.
  - Will be changing defaults so that more code benefits.

Systems Programming

Stack Allocation, Alias Tracking, Byrefs, Fast Delegates and Interfaces

Async Model

- Methods declare their ability to introduce interleaving.

```
awaits Client Connect(IPv4Address address)
{
    Socket socket = await Socket.Connect(address);
    return new Client(socket);
}
```

- Illegal to call awaiting code from non-awaiting code; spawn a task instead.

```
AsyncFactory factory = …;
var result = factory.Run Connect(address);
```

```
await AsyncFactory.WaitAll(
    () => await Connect(address1)
    () => await Connect(address2)
);
```

- Began with a CPS-style translation a la C# 5.0.
  - Transitioned some code and techniques back to .NET.
  - Unfortunately, quite GC heavy; did not meet our performance aspirations.

- Implemented lightweight coroutines with linked stacks. Zero-alloc 'awaits' methods.

Byref Returns, Fast Interfaces and Delegates

- Byref returns and locals.

```
ref T GetAt(int index)
{
    return ref m_array[index];
}
```

- Only possible thanks to race-free programming model (no tearing):
  - Fat interfaces.
    - Two words: object ref, itable pointer.
    - Interface invoke is faster than virtual invoke!

CLR Virtual Call	CLR Interface Invoke	Midori Virtual Call	Midori Interface Invoke
116.26%	147.00%	100.32%	100.00%

  - Value type delegates.
    - Two-word structs.
    - Combined with scoped delegates, zero-alloc lambdas.

Stack Allocation and Scoping

- Allow programmers to control aliasing and lifetime.
- Make stack/embedded allocation safe, easy, and commonplace.

```
class List<T>
{
    …
    public void Contains(scoped Func<T, bool> predicate) scoped;
}

val List<int> myList = new List<int>() { … };

scoped List<int> myListAlias = myList;

int v = 42;

if (myListAlias.Contains(x => x == v)) {
    …
}
```

Byref Returns, Fast Interfaces and Delegates

- Byref returns and locals.

```
ref T GetAt(int index)
{
    return ref m_array[index];
}
```

- Only possible thanks to race-free programming model (no tearing):
  - Fat interfaces.
    - Two words: object ref, itable pointer.
    - Interface invoke is faster than virtual invoke!

CLR Virtual Call	CLR Interface Invoke	Midori Virtual Call	Midori Interface Invoke
116.26%	147.00%	100.32%	100.00%

  - Value type delegates.
    - Two-word structs.
    - Combined with scoped delegates, zero-alloc lambdas.

Async Model

- Methods declare their ability to introduce interleaving.

```
awaits Client Connect(IPv4Address address)
{
    Socket socket = await Socket.Connect(address);
    return new Client(socket);
}
```

- Illegal to call awaiting code from non-awaiting code; spawn a task instead.

```
AsyncFactory factory = …;
var result = factory.Run Connect(address);
```

```
await AsyncFactory.WaitAll(
    () => await Connect(address1)
    () => await Connect(address2)
);
```

- Began with a CPS-style translation a la C# 5.0.
  - Transitioned some code and techniques back to .NET.
  - Unfortunately, quite GC heavy; did not meet our performance aspirations.

- Implemented lightweight coroutines with linked stacks. Zero-alloc 'awaits' methods.

Byref Returns, Fast Interfaces and Delegates

- Byref returns and locals.

```
ref T GetAt(int index)
{
    return ref m_array[index];
}
```

- Only possible thanks to race-free programming model (no tearing):
    - Fat interfaces.
        - Two words: object ref, itable pointer.
        - Interface invoke is faster than virtual invoke!

CLR Virtual Call	CLR Interface Invoke	Midori Virtual Call	Midori Interface Invoke
116.26%	147.00%	100.32%	100.00%

    - Value type delegates.
        - Two-word structs.
        - Combined with scoped delegates, zero-alloc lambdas.

Stack Allocation and Scoping

- Allow programmers to control aliasing and lifetime.
- Make stack/embedded allocation safe, easy, and commonplace.

```
class List<T>
{
    …
    public void Contains(scoped Func<T, bool> predicate) scoped;
}

val List<int> myList = new List<int>() { … };

scoped List<int> myListAlias = myList;

int v = 42;

if (myListAlias.Contains(x => x == v)) {
    …
}
```

Deterministic Destruction

```
class SecureData
{
    byte[] m_data = …;

    ~SecureData()
    {
        m_data.Clear();
    }
}

val SecureData data = new SecureData();
scoped SecureData scopedData = data;
…
// data.~dtor() is implicitly called here;
// ‘scopedData’ is no longer usable
```

Midori's Error Philosophy

- Fail fast on coding errors.
  - Null dereferences.
  - Array out-of-bounds.
  - Overflows (checked arithmetic is the default).
- Minimize dynamic and recoverable failures.
  - Usually a bug farm – not well tested.
  - 90% of exceptions in .NET represent coding errors; see above.
  - But they are still supported; e.g., parsers, reactive evaluations, I/O.
- Make it easy to find bugs fast.
- Allow the compiler to aggressively optimize; no-throw ⇒ less EH bloat.
- Tried many models over 3 years; is now working great in practice (stable for a year).

Contracts and Assertions

- Contracts are fail-fast, subtyping-friendly, and side-effect-free.

```
virtual string Format(int x)
    requires x > 0
    ensures return != null
{
    if (s == 1) {
        return "One";
    }
    else {
        return "More than one";
    }
}
```

```
override string Format(int x)
    requires base
    ensures base
{
    …
    Debug.Assert(IsValid());
    …
}

private bool IsValid() readable
{
    …
}
```

Recoverable Errors: Exceptions

- Methods cannot throw by default.
  - Predictability, great CQ.
  - >90% of our methods are guaranteed not to throw.
- If a method can throw, callers and callee must opt-in.
  - Great for readability – obvious precisely what throws.

```
int Parse(string s)
{
    if (s == "One") {
        return 1;
    }
    else {
        throw new Exception(); // error
    }
}
```

```
int Method(string s)
requires s != null
{
    Parse(s);
}
```

Typed Exceptions

- Most methods have a single failure mode.
  - Must opt-in to depend on Exception types (statically, not dynamically).

```
throws int ParseCore(string s)

{
    if (s == "One") {
        return 1;
    }
    else {
        throw new MyException("Invalid input");
    }
}

int Parse()
{
    try {
        try ParseCore("One");    // error CS7303: An exception of type 'System.Exception' may be thrown from
                                 // this expression; either handle this exception, or make the enclosing
                                 // method 'throws'

    }
    catch (MyException ex) {    // warning CS7335: Unreachable exception handler detected; nothing in this 'try'
                                // block throws an exception of type 'MyException'
        …
    }
}
```

Recoverable Errors: Exceptions

- Methods cannot throw by default.
  - Predictability, great CQ.
  - >90% of our methods are guaranteed not to throw.
- If a method can throw, callers and callee must opt-in.
  - Great for readability – obvious precisely what throws.

```
throws int Parse(string s)
{
    if (s == "One") {
        return 1;
    }
    else {
        throw new Exception();
    }
}
```

```
throws int Method(string s)
    requires s != null
{
    try Parse(s);
}
```

Typed Exceptions

- Most methods have a single failure mode.
  - Must opt-in to depend on Exception types (statically, not dynamically).

```
throws int ParseCore(string s)

{
    if (s == "One") {
        return 1;
    }
    else {
        throw new MyException("Invalid input");
    }
}

int Parse()
{
    try {
        try ParseCore("One");    // error CS7303: An exception of type 'System.Exception' may be thrown from
                                 // this expression; either handle this exception, or make the enclosing
                                 // method 'throws'

    }
    catch (MyException ex) {    // warning CS7335: Unreachable exception handler detected; nothing in this 'try'
                                // block throws an exception of type 'MyException'
        …
    }
}
```

What’s Left

- The Big One: non-null types (by default).
- Changing the defaults – principle of least authority:
  - **readonly** fields by default; **mutable** opts out
  - byval capture in delegates by default; **ref** opts out
  - **sealed** types by default; **virtual** opts out
  - **readable** the default for references (controversial!)
  - **scoped** the default for ‘this’ parameters (*perhaps* more)
  - “Transparent permissions” for methods and property getters
- Many other small features: e.g., syntactic sugar, void in generics (Func<int, void>).
- Developing guidance for using features; ensuring Midori Framework complies.
- Quite a bit on the cut list; optimizing for “game changers” that are hard to add in V2.

Wrapping Up

- Midori is in production.
  - Real customer value, saving money for Bing.
  - Midori C# is stable, truly used, and headed to V1 this year; Midori Framework soon to follow.

- More big things to come. You'll be hearing about them:
  - Midori Overview MSR talk in March.
  - More internal workloads throughout 2013.

- We always have positions for great people who love to code!

- Feedback appreciated.
  - More details available at http://midori/.
  - OOPSLA paper from last year: http://research.microsoft.com/apps/pubs/default.aspx?id=170528

- Thanks!

  -- Joe, Jared, Aleks, and the Midori Language Design Team